CSE 30321 – Lecture 02-03 – In Class Example Handout

Part A: Discussion – Overview of Stored Programs

Let's walk through what happens so that code you write is eventually executed on-chip.

- We'll use a *stored program model* as context.

After you write code, you *compile* it – i.e. translate it into a sequence of operations that computer HW can understand

- Notes:
  - You can compile the same C-program on different processor architectures (i.e. Intel vs. AMD) and it will work
  - Usually, a layer of compiler different depending on machine target

By compiling, might take a statement of C-code like…

```
x = a * b + c / d * e + f;
```

…and translate it to a somewhat simpler set of "commands":

```
t1 = a * b
t2 = c / d
t3 = t2 * e
t4 = t1 + t3
t5 = t4 + f
```

Simpler commands/operations are called *instructions*

- Instructions represent something the processor HW can actually do.
  - Example:
    - Common DSP function is a multiply-accumulate operation: x ← x + (y*z)
    - Some DSP processors have HW that can do this directly
    - Others may require two separate steps:
      - temp = y * z
      - x = x + temp
      - (thus, there would not be a multiply-accumulate instruction)
- Instructions inherently suggest a sequence of 1s and 0s that the processor can understand

Common instruction syntax looks something like this:

```
t4 = t1 + t3    =    Add t4, t1, t3        # t4 ← t1 + t3
t3 = t2 * e     =    Mult t3, t2, e        # t3 ← t2 * e
```

The tX variables used in the above example are really just random placeholders

- Above code does the exact same thing if: **t3** = moon_unit and **t2** = dweezle

In HW, commonly used data often resides physically in *registers*; registers are close to processing logic and can be accessed quickly.

- Registers are essentially sequences of N flip-flops (refer back to logic design)
- Usually, processor has a finite number, so it's important that a compiler uses them smartly
  - So, an instruction like Add t4, t1, t3 might instead be written as:
    - Add R4, R1, R3
    - Add $4, $1, $3 # $ is a commonly used symbol for register
    - (Instruction tells you that the compiler has assigned certain variables of your program to registers 1, 3, and 4)

Looking back at our initial example, the instructions that comprise this statement might be as follows:

t1 = a * b	Mult R10, R11, R12	R10 = t1; R11 = a; R12 = b
t2 = c / d	Div R13, R14, R15	R13 = t2; R14 = c; R15 = d
t3 = t2 * e	Mult R16, R13, R17	R16 = t3; R17 = e; use R13 data
t4 = t1 + t3	Add R18, R10, R16	R18 = t4; use R10, R16 data
t5 = t4 + f	Add R19, R18, R20	R19 = t5; R20 = f; use R18 data

Now, let's go 1 step further…

- We'll use "Add R18, R10, R16" as an example…
  - This is really just another way of representing a string of 1s and 0s that computer HW can understand
    - Might be: 00101 ■ 10010 ■ 01010 ■ 10000
  - When processor HW is presented with sequence of 1s and 0s it knows to interpret certain parts of the bit sequence in certain ways … i.e.:
    - Last 5 bits are the name/location of one "source operand" (i.e. $10000_2 = 16_{10}$)
    - 2^{nd} to last 5 bits are the name of another source operand (i.e. $01010_2 = 10_{10}$)
    - Another sequence of 5 bits tells processor HW where to store results (i.e. $10010_2 = 18_{10}$)
    - Most significant bits might be an "operation code"
      - Usually every instruction has this field
      - Tells processor HW what to do: i.e. 00100 = "Add"
  - If try to run binary executable on wrong architecture, get errors b/c sequences of 1s and 0s are misinterpreted.

So, our sequence of instructions that we created above might really look like this:

- Note that I've arbitrarily made up "op codes" for the multiply, add, and divide instructions

Instruction mnemoic	Op code	Destination	Source 1	Source 2
Mult R10, R11, R12	00110	01010	01011	01100
Div R13, R14, R15	00111	01101	01110	01111
Mult R16, R13, R17	00110	10000	01101	10001
Add R18, R10, R16	00101	10010	01010	10000
Add R19, R18, R20	00101	10011	10010	10100

When you compile a program, this is what you effectively produce:

- A sequence of instructions that can be processed by your HW
- That sequence of instructions corresponds to a sequence of 1s and 0s that the HW can interpret

So to foreshadow a bit:

- If each instruction takes the same amount of time, a compiler that produces fewer instructions is probably better than 1 that produces more (at least for that particular program)
- Or, if you can do *more* with the same instruction (in the same amount of time), that's a good thing too!

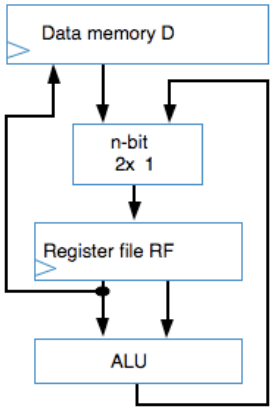
Part B: Discussion – What's a Clock Cycle?

- Recall for logic design you can store a bit of data in a flip-flop
  - In context of architecture, can think of a clock cycle as:
    - Time data in a flip flop … to be processed / move through some combinational logic (i.e. computation) … and to be stored again in another (or the same flip flop)

Part C: Datapath examples

Question 1:

Which are valid, *single cycle operations* for a given datapath? A datapath schematic is included for your reference.



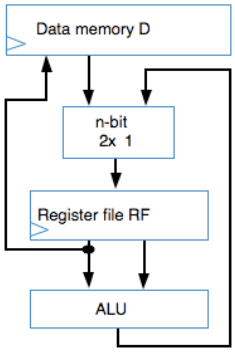
(a) Move D[1] to RF[1] (i.e., RF[1] = D[1])
Yes: this is a load.
Data goes from Data memory → n-bit 2x1 mux → register file

(b) Store RF[1] to D[9] and store RF[2] to D[10]
No: HW not available to do this.
There is only 1 path to the data memory
How would we do this?

(c) Add D[0] plus D[1], store result in D[9]
No: look at where data into the ALU comes from … RF. Also, no path from ALU to data memory.

Question 2:

How many cycles does each of the following take given the above data path. Again, a schematic is included for your reference.



(a) Move RF[1] to RF[2]. (How is this done?)
1st question … HOW? (Not uncommon for there to be default 0.)
Usually, R0 = 0. Therefore, might do RF[2] = RF[1] + RF[0]

(b) Add D[8] with RF[2] and store the result in RF[4].
Syntax: RF[4] = D[8] + RF[2] (this is what we *want* to do)
(i) Load D[8] into register – RF(3) = D[8]; (ii) do add: RF[4] = RF[1] + RF[2]

(c) Add D[8] with RF[1], then add the result with RF[4], and store the result in D[8].
Hint: temp = RF[1] + D[8]; temp2 = temp + RF[4]; D[8] = temp
Ans: **1.** RF[x] = D[8]; **2.** RF[x] = RF[1] + RF[x]; **3.** RF[x] = RF[x] + RF[4]; **4.** D[8] = RF[x]

Question 3:

A. Create a sequence of instructions for the following operations:

D[8] = D[8] + RF[1] + RF[4]

Your answer		My answer	
1.	RF[2] = RF[1] + RF[4]	**3.**	RF[2] = RF[2] + RF[3]
2.	RF[3] = D[8]	**4.**	D[8] = RF[2]

B. Translate your answers in Question 4 into assembly code.

Your answer		My answer	
1.	Add R2, R1, R4	**3.**	Add R2, R2, R3
2.	Mov R3, 8	**4.**	Mov 8, R2

- In 2: (a) Mov could just as easily say "Load"; (b) R3 is the *destination* (1st) and 8 is the *source* address (2nd)
- In 4: (a) Mov could say "Store"; (b) 8 is the destination (1st) and R2 is the *destination* register (2nd)

Part D: Programs for a 6-instruction processor

Question 1:

- Problem:
  - Count number of non-zero words in D[4] and D[5]
  - Result will be either 0, 1, or 2
  - Put result in D[9]

- There are 2 ways to think about doing this problem…
  - Assume "count" initially 2, go down
  - Assume "count" initially 0, go up.

- Let's assume we count down instead of up…

MOV Rx, 2	Load / set count to 2	(Use load constant instruction)
MOV R1, 1	Load #1 into R1	(Use load constant instruction)
MOV R2, 4	Load d(4) into R2	(Use load instruction)
JUMPZ R2, A	If zero, skip next instruction	(Jump if 0 instruction)
Sub Rx, Rx, R1	Update count, word = non zero	(Subtract instruction)
MOV R2, 5	Load d(5) into R3	(Load instruction)
JUMPZ R2, A	If zero, skip next instruction	(Jump if 0 instruction)
Sub Rx, Rx, R1	Update constant	(Subtract instruction)
MOV 9, Rx	Move "count" to d(9)	(Store instruction)

- Now, let's assume that we want to count up instead of down…

	MOV R0, #0	Initialize the result to 0	Load constant
	MOV R1, #1	Load constant 1 into R1 (use for incrementing result)	Load constant
	MOV R2, 4	Get data memory location 4	Load
	JUMPZ R2, p	If zero, skip the next instruction	Jump if zero
	Add R0, R0, R1	Non zero, so increment result	Add
p	MOV R2, 5	Get data memory location 5	Load
	JMPZ R2, q	If zero, skip next instruction	Jump if zero
	Add R0, R0, R1	Non zero, so increment result	Add
q	MOV 9, R0	Store result in data memory location 9	Store

Part E: Setting Control Signals and Modifying the Datapath

Setting the Control Signals

For **Load**:

- D_addr = d
  - Address sent to data memory (to read from) ("/8" = 8 bits)
- D_rd = 1
  - Signal indicates we are doing a memory read
- RF_s = 1
  - Take data from memory, not ALU to load / send to register file
- RF_W_addr
  - The # of the register to load data to; again, how may registers?
- RF_W_wr = 1
  - We are doing a register write

For **Store**:

- D_addr = d
  - Address sent to data memory (to write to)
- D_wr = 1
  - Doing memory write
- DF_s = X
  - We don't care what is selected
- RF_Rp_addr = Ra
  - 4 bits indicating what register data we will read from
- RF_Rp_read = 1
  - We're doing a read, so don't write to register file
  - Important to note that (garbage) data *could* be written to data file

For **Add**:

- RF_RP_addr = rb (4 bit field)
  - 1 address sent to register file to read
- RF_RP_rd = 1
  - Read from the register file
- RF_s = 0
  - Tells which input of the multiplexor should be selected as input to the register file
- RF_Rq_addr = rc (4 bit field)
  - 2^{nd} address sent to the register file to read
- RF_Rq_rd = 1
  - Read from the register file
- RF_W_addr = ra
  - Send specific 4 bits of the IR to the register file – indicates what register will be written
- RF_W_wr = 1
  - Set a flag indicating it is OK to write to that regsiter
- ALU_s0 = 1
  - Tells the ALU what function to perform – e.g. add or subtract…
    - OR vs. Add example…

Extending the Control Unit and Datapath

1. The *load constant* instruction…
   a. …requires that the register file be able to load data from *IR[7..0]*
   b. …in addition to data from data memory or the ALU output; thus, we widen the register file's multiplexer from 2x1 to 3x1, add another mux control signal, and also create a new signal coming from the controller labeled *RF_W_data*, which will connect with *IR[7..0]*.

2. The subtract instruction requires that we use an ALU capable of subtraction, so we add another ALU control signal.

3. The jump-if-zero instruction requires that we be able to detect if a register is zero, and that we be able to add *IR[7..0]* to the *PC*.
   a. We insert a datapath component to detect if the register file's *Rp* read port is all zeros (that component would just be a NOR gate).
   b. We also upgrade the *PC* register so it can be loaded with *PC* plus *IR[7..0]*. The adder used for this also subtracts 1 from the sum, to compensate for the fact that the *Fetch* state already added 1 to the *PC*.

Part F: Design Level Examples

Question 1:

Assume that you have the following piece of C-code:

```
for (i=1; i<5; i++) {
        a = a + a;
}
b = a;
```

Part A:

Assuming the instruction set associated with the 6-instruction processor, translate this C-code to machine instructions. You should assume that a = R1 and b = d(9). (You'll need to make a few other assumptions too, but this is for you to figure out.)

Start with pseudo-code:

- Load max loop count
- Load constant to update count
- ** Perform Add (note – only "computation")
- Update counter
- Check counter value
  - If 0, store result
  - If not 0, go back to **
- Store result

Machine code:

```
        MOV   R2, #4          // Use load constant; initialize to 4 b/c of < 5, not <= 5
        MOV   R3, #1          // Use load constant; put #1 in register
Y:      ADD   R1, R1, R1      // Perform add; use add
        SUB   R2, R2, R3      // Decrement counter; use subtract result
        JUMPZ R2, X           // Goto X if counter = 0 (we're done…)
        JUMPZ R0, Y           // If not, 0, need to do again (R0 == 0)
                                 (y must be negative; PC = PC – offset)
X:      MOV 9, R1             // Store result
```

Question 2:

Part A:
In this question, we're going to work with the initial datapath for the 3-instruction processor (shown below). Let's assume that we want to modify this datapath to add an instruction that performs a "add immediate" operation. In terms of register transfer language (RTL), describe what the instruction should do during the execute stage:

Rx ← Ra + IR(3:0)

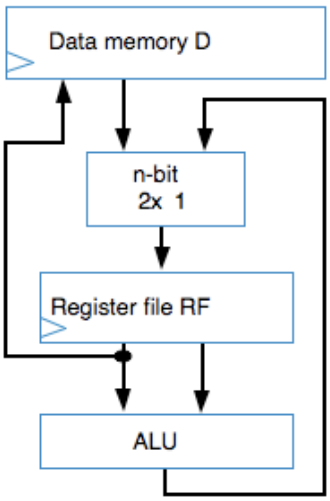
Thus, if we assume that the opcode for this new instruction (let's call it ADDi) is 1101, then a sample bit encoding might look something like this:

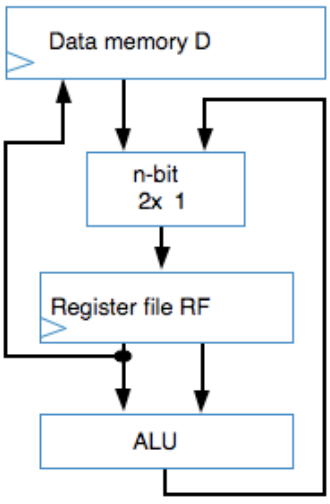
1101 # 0101 # 0111 # 0001

This instruction would do the following:

R5 ← R7 + 1

Part B:
Modify the datapath so that we can successfully execute the new ADDi instruction:





(spare)

Part C:
Using the new datapath you've designed, write out the machine instructions for performing the sequence of operations:

```
a = b + c;                  // A = R1, B = R2, C = R3
c = c + 12;                 //
b = a + 1;
d = b + 19;                 // D = R4
```

Answer:

```
ADD   R3, R1, R2
ADDi  R3, R3, #12
ADDi  R2, R1, #1
MOV   R5, 19
ADD   R4, R2, R5
```

Part D:
For comparison, let's write out the sequence of instructions to perform the same sequence of operations without the new instruction. *Quantify* how much the new instruction helps.

```
ADD   R3, R1, R2
MOV   R7, #12
ADD   R3, R3, R7
MOV   R8, #1
ADD   R2, R1, R8
MOV   R5, #19
ADD   R4, R2, R5
```

Each instruction takes 3 cycles. From Part C, 5 instructions are required with the ADDi instruction. Without the ADDi instruction, 7 instructions are required. Thus, if each instruction takes 3 CCs, then the solution from Part D will be take (21/15 = 1.4) 40% longer than the solution in Part C. For the amount of hardware required to implement it, this would probably be a pretty good design decision.